4

Technical Document 1893
July 1990

Models of Software Evolution

Life Cycle and Process

University of Southern California

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This work was performed for the Naval Ocean Systems Center, San Diego, CA 92152-5000, under program element 0602234N. Contract N66001-87-D-0179 was carried out by the University of Southern California, Computer Science Department, Los Angeles, CA 90089-0782, under the technical coordination of L. Sutton, Computer Systems and Software Technology Branch, Code 411, Naval Ocean Systems Center.

Released by
D. L. Hayward, Head
Computer Systems and
Software Technology Branch

Under authority of
A. G. Justice, Head
Information Processing
and Displaying Division

FS

CONTENTS

FIGURES

1.0 INTRODUCTION

Software evolution represents the cycle of activities involved in the development, use, and maintenance of software systems. Software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. In this paper, we categorize and examine a number of schemes for modelling software evolution. We start with some definitions of the terms used to characterize and compare different models of software evolution. We next review the traditional software life cycle models which dominate most textbook discussions and current software development practices. This is followed by a more comprehensive review of the alternative models of software evolution that have been recently proposed and used as the basis for organizing software engineering projects and technologies. As such, we then examine what are the role of existing and emerging software engineering technologies in these models. We then provide some practical guidelines for evaluating the alternative models of software evolution, and for customizing an evolutionary model to best suit your needs. Ultimately, the objective in this paper is to assess our ability to articulate the basis for substantive theory of software evolution which can serve as both a guide for organizing software development efforts as well as a basis for organizing empirical studies that can test, validate, and refine hypotheses about the statics and dynamics of software evolution.

We start this examination with some background definitions and concepts.

1.1 BACKGROUND

Charles Darwin is often recognized as the person who focused scientific attention to the problems of developing an empirically-grounded theory of biological evolution. Darwin's model of the evolution of species was provocative in both scientific and religious circles. His model identifies four central characteristics that account for the development of natural species: (a) emergence of variations within existing species, (b) some mechanism of inheritance to preserve the variations, (c) tendency to multiply (reproduce) leading to competition for scarce resources, and (d) environmentally-driven "natural" selection across generations.

Darwin's model was monumental in precipitating a realignment in research methods and theories in the biological sciences, establishing a separate discipline of evolutionary biology, and giving rise to thousands of experiments that sought to refute, validate, or extend his empirically grounded theori⁻s and conjectured hypotheses of biological evolution [22]. The concept of what we now identify as "system life cycle" thus has in its historical origins in the field of evolutionary biology. The field oı cybernetics however added the control and evolution of technological systems to the evolutionar⁻ liie cycle concept. In turn, the notions of software life cycle and evolution we examine in this pap·r can be traced back to these scientific roots.

Nonetheless, we should recognize that the emergence of a viable theory of software evolution might lead to paradigmatic shifts similar to those put in to motion by Darwin's ·heory. Alternatively, a viable theory of software evolution might lead to new insights that shed light on how to most effectively resolve some of the long-standing dilemmas of software engineering including how to: improve development productivity, reduce development and maintenance costs, better facilitate improved software project management, streamline the software technology transfe: process, and shape the funding and policy objectives for national software engineering research initiatives. In some sense, these are the ultimate "reality check" or validation criteria that a theory of software evolution will be subjected to.

Explicit models of software evolution date back to the earliest projects developing large software systems in the 1950's and 1960's [15, 44, 74]. However, in contrast to the inter-generational Darwinian model, software system development was cast, and generally still remains, as an intra-generational process of growth and maturation.

Overall, the apparent purpose of these early software life cycle models was to provide an abstract scheme for rationally managing the development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities. Figure 1 shows a flow-chart diagram of one of the first software development life cycle models published in 1961 [44].

1.2 SOFTWARE LIFE CYCLE ACTIVITIES

For more than three decades, many descriptions of the classic software life cycle (often referred to as "the waterfall chart") have appeared (e.g., [15, 74, 17, 31, 77, 33]). See Figure 2 for an example of such a chart. These charts are often employed during introductory presentations to people who may by unfamiliar with what kinds of technical problems and strategies must be addressed when constructing large software systems.

These classic software life cycle models usually include some version or subset of the following activities:

- System Initiation/Adoption: where do systems come from? In most situations, new systems replace or supplement existing information processing mechanisms whether they were previously automated, manual, or informal.
- Requirement Analysis and Specification: identifies the problems a new software system is supposed to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance.
- Functional Specification or Prototyping: identifies and potentially formalizes the objects of computation, their attributes and relationships, the operations that transform these objects, the constraints that restrict system behavior, and so forth.
- Partition and Selection (Build vs. Buy vs. Reuse): given requirements and functional specifications, divide the system into manageable pieces that denote logical subsystems, then determine whether new, existing, or reusable software systems correspond to the needed pieces.
- Architectural Configuration Specification: defines the interconnection and resource interfaces between system modules in ways suitable for their detailed design and overall configuration management.
- Detailed Component Design Specification: defines the procedural methods through which each module's data resources are transformed from required inputs into provided outputs.
- Component Implementation and Debugging: codifies the preceding specifications into operational source code implementations and validates their basic operation.
- Software Integration and Testing: affirms and sustains the overall integrity of the software system architectural configuration through verifying the consistency and completeness of implemented modules, verifying the resource interfaces and interconnections against their specifications, and validating the performance of the system and subsystems against their requirements.

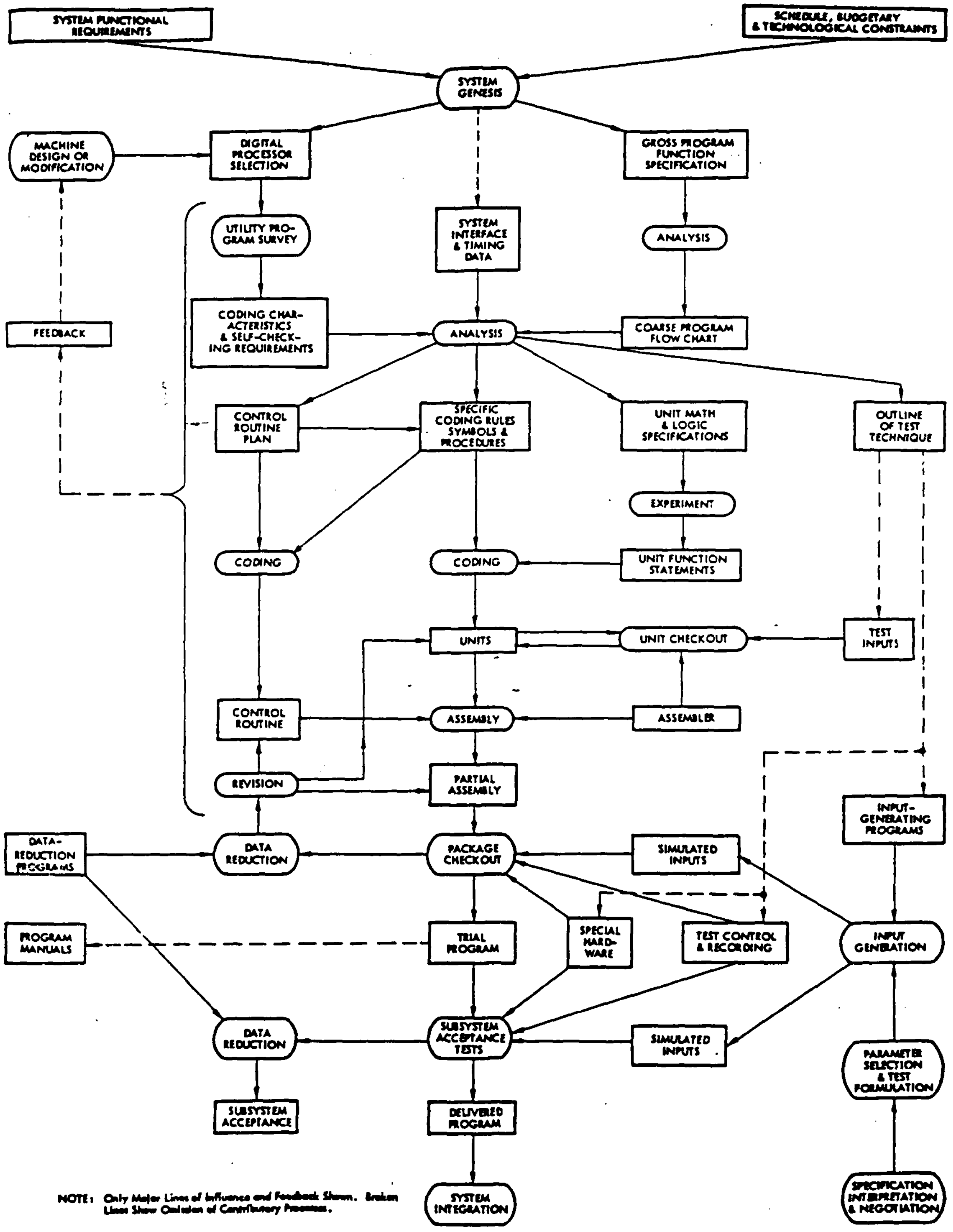
SYSTEM FUNCTIONAL REQUIREMENTS
SCHEDULE, BUDGETARY & TECHNOLOGICAL CONSTRAINTS
SYSTEM GENESIS
MACHINE DESIGN OR MODIFICATION
DIGITAL PROCESSOR SELECTION
GROSS PROGRAM FUNCTION SPECIFICATION
UTILITY PROGRAM SURVEY
SYSTEM INTERFACE & TIMING DATA
ANALYSIS
FEEDBACK
CODING CHARACTERISTICS & SELF-CHECKING REQUIREMENTS
ANALYSIS
COARSE PROGRAM FLOW CHART
CONTROL ROUTINE PLAN
SPECIFIC CODING RULES SYMBOLS & PROCEDURES
UNIT MATH & LOGIC SPECIFICATIONS
OUTLINE OF TEST TECHNIQUE
EXPERIMENT
CODING
CODING
UNIT FUNCTION STATEMENTS
UNITS
UNIT CHECKOUT
TEST INPUTS
CONTROL ROUTINE
ASSEMBLY
ASSEMBLER
REVISION
PARTIAL ASSEMBLY
INPUT-GENERATING PROGRAMS
DATA-REDUCTION PROGRAMS
DATA REDUCTION
PACKAGE CHECKOUT
SIMULATED INPUTS
PROGRAM MANUALS
TRIAL PROGRAM
SPECIAL HARDWARE
TEST CONTROL & RECORDING
INPUT GENERATION
DATA REDUCTION
SUBSYSTEM ACCEPTANCE TESTS
SIMULATED INPUTS
PARAMETER SELECTION & TEST FORMULATION
SUBSYSTEM ACCEPTANCE
DELIVERED PROGRAM
SYSTEM INTEGRATION
SPECIFICATION INTERPRETATION & NEGOTIATION
NOTE: Only Major Lines of Influence and Feedback Shown. Broken Lines Show Omission of Contributory Processes.

Figure 1. An early software life cycle model, from W. A. Hosier.

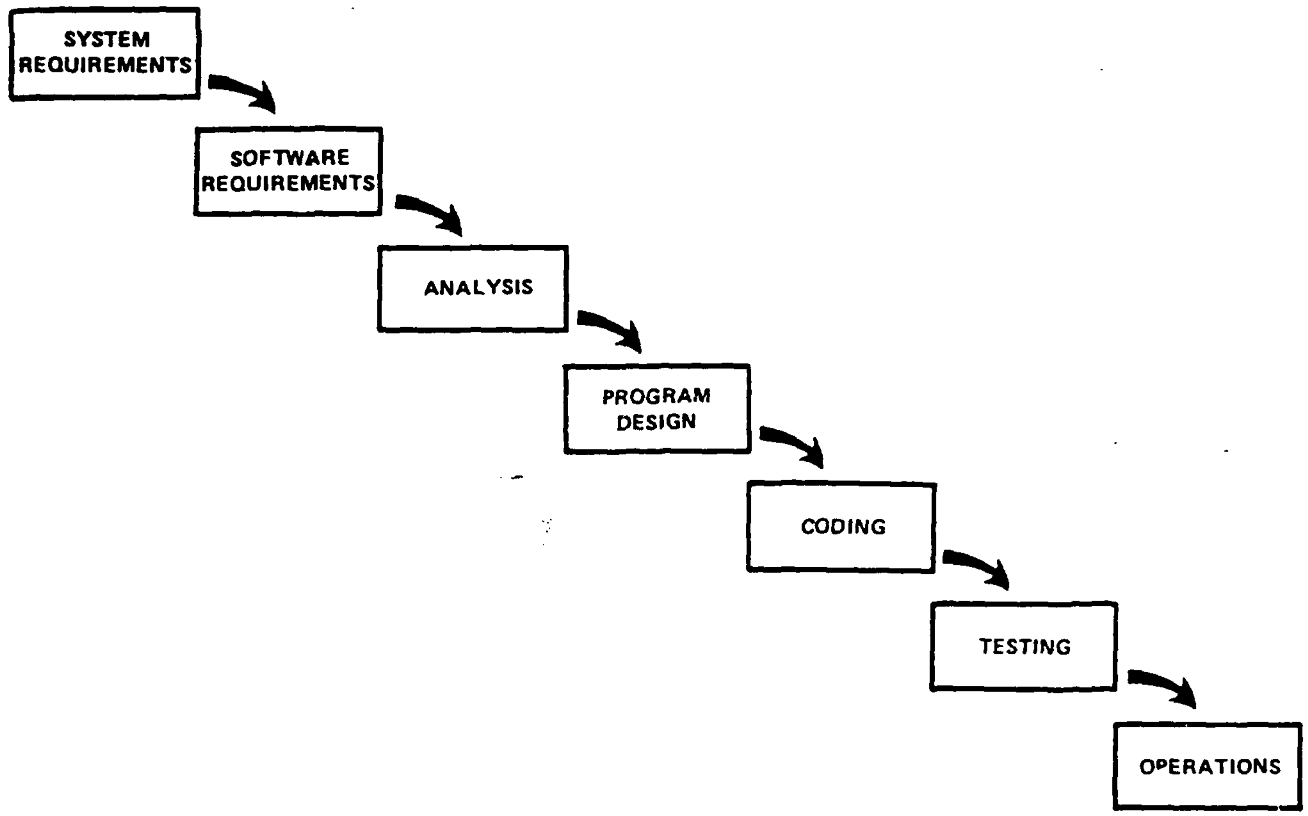

Figure 2. A software life cycle "waterfall chart."

- Documentation Revision and System Delivery: packaging and rationalizing recorded system development descriptions into systematic documents and user guides, all in a form suitable for dissemination and system support.
- Deployment and Installation: providing directions for installing the delivered software into the local computing environment, configuring operating systems parameters and user access privileges, running diagnostic test cases to assure the viability of basic system operation.
- Training and Use: providing system users with instructional aids and guidance for understanding the system's capabilities and limits in order to effectively use the system.
- Software Maintenance: sustaining the useful operation of a system in its host/target environment by providing requested functional enhancements, repairs, performance improvements, and conversions.

1.3 WHAT IS A SOFTWARE LIFE CYCLE MODEL?

A software life cycle model is either a descriptive or prescriptive characterization of software evolution. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well-reasoned. In turn, this allows these models to be used as a basis for software project organization. This means that many idiosyncratic details for how to organize a software development effort can be ignored, glossed over, generalized, or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models when developing different kinds of application systems, in different kinds of development settings, using different programming languages, with differentially skilled staff, etc. However, prescriptive models are also used to package the development tasks and techniques for using a given set of software engineering tools or environment during a development project.

Descriptive life cycle models, on the other hand, characterize how particular software systems are actually developed in specific settings. As such, they are less common and more difficult to articulate for an obvious reason: one must observe or collect data throughout the life cycle of a software system, a period of elapsed time usually measured in years. Also, descriptive models are specific to the systems observed, and only generalizable through systematic comparative analysis. Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.

These two characterizations suggest that there are a variety of purposes for articulating software life cycle models. This variety includes:

- To organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
- As prescriptive outlines for what documents to produce for delivery to client.
- As a basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities.
- As frameworks for analyzing or estimating patterns of resource allocation and consumption during the software life cycle [18].
- As comparative descriptive or prescriptive accounts for how software systems come to be the way they are.
- As a basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

1.4 WHAT IS A SOFTWARE PROCESS MODEL?

A software process model often represents a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution [71, 88, 32]. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing.

Software process networks can be viewed as representing multiple interconnected task chains [56, 34]. Task chains represent a non-linear sequence of actions {By this we mean that the sequence of actions may be nondeterministic, iterative, accommodate multiple/parallel alternatives, as well as be partially ordered to account for incremental progress.} that structure and transform available computational objects (resources) into intermediate or finished products. Task actions in turn can be viewed as non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard. Winograd and others have referred to these units of cooperative work between people and computers as "structured discourses of work" [90].

Task chains can be employed to characterize either prescriptive or descriptive action sequences. Prescriptive task chains are idealized plans of what actions should be accomplished, and in what order. For example, a task chain for the activity of object-oriented software design might include the following task actions:

- Develop an informal narrative specification of the system.
- Identify the objects and their attributes.
- Identify the operations on the objects.
- Identify the interfaces between objects, attributes, or operations.
- Implement the operations.

Clearly, this sequence of actions could entail multiple iterations and non-procedural primitive action invocations in the course of incrementally progressing toward an object-oriented software design.

Task chains join or split into other task chains resulting in an overall production lattice [56]. The production lattice represents the "organizational production system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems. The production lattice therefore structures how a software system is developed, used, and maintained. However, prescriptive tasks chains and actions cannot be formally guaranteed to anticipate all possible circumstances or idiosyncratic foul-ups that can emerge in the real-world of software development [37, 36]. Thus any software production lattice will in some way realize only an approximate or incomplete description of software development. As such, articulation work {Articulation work in the context of software evolution includes actions people take that entail either their accommodation to the contingent or anomalous behavior of a software system, or negotiation with others who may be able to affect a system modification or otherwise alter current circumstances [14]. In other places, this notion of articulation work has been referred to as software process dynamism.} will be performed when a planned task chain is inadequate or breaks down. The articulation work can then represent an open-ended nondeterministic sequence of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain [14]. Thus, descriptive task chains are employed to characterize the observed course of events and situations that emerge when people try to follow a planned task sequence.

1.5 EVOLUTIONISTIC VS. EVOLUTIONARY MODELS

Every model of software evolution makes certain assumptions about what is the meaning of evolution. In one such analysis of these assumptions, two distinct views are apparent: evolutionistic models focus attention to the direction of change in terms of progress through a series of stages eventually leading to some final stage; evolutionary models on the other hand focus attention to the mechanisms and processes that change systems [54]. Evolutionistic models are often intuitive and useful as organizing frameworks for managing and tooling software development efforts. But they are poor predictors of why certain changes are made to a system, and why systems evolve in similar or different ways [14]. Evolutionary models are concerned less with the stage of development, but more with the technological mechanisms and organizational processes that guide the emergence of a system over space and time. As such, it should become apparent that the prescriptive models are typically evolutionistic, while most of the alternative models are evolutionary.

1.6 THE NEGLECTED ACTIVITIES OF SOFTWARE EVOLUTION

Three activities critical to the overall evolution of software systems are maintenance, technology transfer, and evaluation. However, these activities are often inadequately addressed in most models of software evolution. Thus, any model of software evolution should be examined to see to what extent it addresses these activities.

Software maintenance often seems to be described as just another activity in the evolution of software. However, many studies indicate that software systems spend most of their useful life in this activity [17, 18]. A reasonable examination of the activity indicates that maintenance represent ongoing incremental iterations through the life cycle or process activities that precede it [8]. These iterations are an effective way to incorporate new functional enhancements, remove errors, restructure code, improve system performance, or convert a system to run in another environment. The various instances of these types of software system alterations emerge through ongoing system use within regular work activities and settings. Subsequently, software maintenance activities represent smaller or lower profile passages through the life cycle. Further, recent interest in software re-engineering and reverse engineering as new approaches to software maintenance suggests that moving forward in a software life cycle depends on being able to cycle backwards [26]. However, it is also clear that many other technical and organizational circumstances profoundly shape the evolution of a software system and its host environment [62, 60, 14]. Thus, every software life cycle or process model should be closely examined to see to what extent it accounts for what happens to a software system during most of its sustained operation.

Concerns for system adoption, installation and support can best be addressed during the earliest stages of software evolution. These entail understanding which users or clients control the resources (budgets, schedules, staff levels, or other "purse strings") necessary to facilitate the smooth transition from the current system in place to the new adopted system. It also entails understanding who will benefit (and when) from smooth software technology transfer and transition, as well as who will be allowed to participate and express their needs. These concerns eventually become the basis for determining the success or failure of software system use and maintenance activities. Early and sustained involvement of users in system development is one of the most direct ways to increase the likelihood of successful software technology transfer. Failure to involve users is one of the most common reasons why system use and maintenance is troublesome. Thus, any model of software evolution can be evaluated according to the extent that it accommodates activities or mechanisms that encourage system developers and users to more effectively cooperate.

Evaluating the evolution of software systems helps determine which development activities or actions could be made more effective. Many models of software evolution do not address how system developers (or users) should evaluate their practices to determine which of their activities could be improved or restructured. For example, technical reviews and software inspections often focus attention to how to improve the quality of the software products being developed, while the organizational and technological processes leading to these products receive less attention. Evaluating development activities also implies that both the analytical skills and tools are available to a development group. Thus, models of software evolution can also be scrutinized to determine to what extent they incorporate or structure development activities in ways that provide developers with the means to evaluate the effectiveness of the engineering practices.

Finally, one important purpose of evaluating local practices for software evolution is to identify opportunities where new technologies can be inserted. In many situations, new software engineering tools, techniques, or management strategies are introduced during the course of a system development effort. How do such introductions impact existing practices? What consequences do such introductions have on the maintainability of systems currently in use or in development? In general, most models of software evolution say little about such questions. However, software maintenance, technology transfer, and process evaluation are each critical to the effective evolution of software systems, as is their effect on each other. Thus, they should be treated collectively, and in turn, models of software evolution can be reviewed in terms of how well they address this collective.

2.0 TRADITIONAL SOFTWARE LIFE CYCLE MODELS

Traditional models of software evolution have been with us since the earliest days of software engineering. In this section, we identify four. The classic software life cycle (or "waterfall chart") and stepwise refinement models are widely instantiated in just about all books on modern programming practices and software engineering. The incremental release model is closely related to industrial practices where it most often occurs. Military standards based models have also reified certain forms of the classic life cycle model into required practice for government contractors. Each of these four models use "coarse-grain" or macroscopic characterizations {The progressive steps of software evolution are often described as "stages"—such as requirements specification, preliminary design, and implementation—which have usually had little or no further characterization other than a list of attributes that the product of such a stage should possess.} when describing software evolution. Further, these models are independent of any organizational development setting, choice of programming language, software application domain, etc. In short, the traditional models are context-free rather than context-sensitive. But as all of these life cycle models have been in use for some time, we refer to them as the traditional models, and characterize each in turn:

2.1 CLASSIC SOFTWARE LIFE CYCLE

The classic software life cycle is often represented as a simple waterfall software phase model, where software evolution proceeds through an orderly sequence of transitions from one phase to the next in order [74]. Such models resemble finite state machine descriptions of software evolution. However, such models have been perhaps most useful in helping to structure, staff, and manage large software development projects in complex organizational settings, which was one of the primary purposes [74, 17]. Alternatively, these classic models have been widely characterized as both poor descriptive and prescriptive models of how software development "in-the-small" or "in-the-large" can or should occur.

2.2 STEPWISE REFINEMENT

In this approach, software systems are developed through the progressive refinement and enhancement of high-level system specifications into source code components [91]. However, the choice and order of which steps to choose and which refinements to apply remain unstated. Instead, formalization is expected to emerge within the heuristics and skills that are acquired and applied through increasingly competent practice. This model has been most effective and widely applied in helping to teach individual programmers how to organize their software development work. Many interpretations of the classic software life cycle thus subsume this approach within their design and implementations.

2.3 ITERATIVE ENHANCEMENT, INCREMENTAL DEVELOPMENT AND RELEASE

Developing systems through incremental release requires first providing essential operating functions, then providing system users with improved and more capable versions of a system at regular intervals [8, 86]. This model combines the classic software life cycle with iterative enhancement at the level of system development organization. It also supports a strategy to periodically distribute software maintenance updates and services to dispersed user communities. This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by many commercial software firms and system vendors. More recently, this approach has been extended through the use of software prototyping tools and techniques (described later), which more directly provide support for incremental development and iterative release for early and ongoing user feedback and evaluation [39]. Last, the Cleanroom software development method at use in IBM and NASA laboratories provides incremental release of software functions and/or subsystems (developed through stepwise refinement) to separate in-house quality assurance teams that apply statistical measures and analyses as the basis for certifying high-quality software systems [82, 65].

2.4 INDUSTRIAL AND MILITARY STANDARD MODELS

Industrial firms often adopt some variation of the classic model as the basis for standardizing their software development practices [74, 17, 31, 77, 78]. Such standardization is often motivated by needs to simplify or eliminate complications that emerge during large software development or project management.

Many government contractors organize their software development activities according to military standards such as that embodied in MIL-STD-2167 [64]. Such standards outline not only a variant of the classic life cycle activities, but also the types of documents required by clients who procure either software systems or complex platforms with embedded software systems. Military software systems are often constrained in ways not found in industrial or academic practice, including: (a) required use of military standard computing equipment (which is often technologically dated and possesses limited proces .ng capabilities); (b) are embedded in larger systems (e.g., airplanes, submarines, missiles, command and control systems) which are "mission-critical" (i.e., those whose untimely failure could result in military disadvantage and/or life-threatening risks); (c) are developed under contract to private firms through cumbersome procurement and acquisition procedures that can be subject to public scrutiny and legislative intervention; and (d) many embedded software systems for the military are among the largest and most complex systems in the world. In a sense, military software standards are applied to simplify and routinize the administrative processing, review, and oversight required by such institutional circumstances. However, this does not guarantee that delivered software systems will be easy to use or maintain. Nor does it necessarily indicate what decisions processes or trade-offs were made in developing the software so as to conform to the standards, to adhere to contract constraints, or to insure attainment of contractor profit margins. Thus, these conditions may not make software development efforts for the military necessarily most effective or well-engineered.

In industrial settings, standard software development models represent often provide explicit detailed guidelines for how to deploy, install, customize or tune a new software system release in its operating application environment. In addition, these standards are intended to be compatible with provision of software quality assurance, configuration management, and independent verification and validation services in a multi-contractor development project. Recent progress in industrial practice appears in [47, 72, 94]. However, neither such progress, nor the existence of such standards within a company, necessarily implies to what degree standards are routinely followed, whether new staff hires are trained in the standards and conformance, or whether the standards are considered effective.

2.5 ALTERNATIVES TO THE TRADITIONAL SOFTWARE LIFE CYCLE MODELS

There are at least three alternative sets of models of software evolution. These models are alternatives to the traditional software life cycle models. These three sets focus of attention to either the products, production processes, or production settings associated with software evolution. Collectively, these alternative models are finer-grained, often detailed to the point of computational formalization, more often empirically grounded, and in some cases address the role of new automated technologies in facilitating software evolution. As these models are not in widespread practice, we examine each set of models in the following sections.

3.0 SOFTWARE PRODUCT DEVELOPMENT MODELS

Software products represent the information-intensive artifacts that are incrementally constructed and iteratively revised through a software development effort. Such efforts can be modeled using software product life cycle models. These product development models represent an evolutionary revision to the traditional software life cycle models. The revisions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, application generators, and documentation support environments. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the software development effort or more rapidly. Therefore in this regard, the models of software evolution may be implicit in the use of the technology, rather than explicitly articulated. This is possible because such models become increasingly intuitive to those developers whose favorable experiences with these technologies substantiates their use. Thus, detailed examination of these models is most appropriate when such technologies are available for use or experimentation.

3.1 RAPID PROTOTYPING

Prototyping is a technique for providing a reduced functionality or a limited performance version of a software system early in its development [3, 84, 20, 23, 40, 27]. In contrast to the classic system life cycle, prototyping is an approach whereby more emphasis, activity, and processing is directed to the early stages of software development (requirements analysis and functional specification). In turn, prototyping can more directly accommodate early user participation in determining, shaping, or evaluating emerging system functionality. As a result, this up-front concentration of effort, together with the use of prototyping technologies, seeks to trade-off or otherwise reduce downstream software design activities and iterations, as well as simplify the software implementation effort.

Software prototypes come in different forms including throwaway prototypes, mock-ups, demonstration systems, quick-and-dirty prototypes, and incremental evolutionary prototypes [27]. Increasing functionality and subsequent ability to evolve is what distinguishes the prototype forms on this list.

Prototyping technologies usually take some form of software functional specifications as their starting point or input, which in turn is either simulated, analyzed, or directly executed. As such, these technologies allow software design activities to be initially skipped or glossed over. In turn, these technologies can allow developers to rapidly construct early or primitive versions of software systems that users can evaluate. These user evaluations can then be incorporated as feedback to refine the emerging system specifications and designs. Further, depending on the prototyping technology, the complete working system can be developed through a continual revising/refining the input specifications. This has the advantage of always providing a working version of the emerging system, while redefining software design and testing activities to input specification refinement and execution. Alternatively, other prototyping approaches are best suited for developing throwaway or demonstration systems, or for building prototypes by reusing part/all of some existing software systems.

3.2 ASSEMBLING REUSABLE COMPONENT SET

The basic approach of reusability is to configure and specialize pre-existing software components into viable application systems [16, 66, 38]. Such source code components might already have associated specifications and designs associated with their implementations, as well as have been tested and certified. However, it is also clear that software specifications, designs, test case suites may themselves be treated as reusable software development components. Therefore, assembling reusable software components is a strategy for decreasing software development effort in ways that are compatible with the traditional life cycle models.

The basic dilemmas encountered with reusable software component set include (a) how to define an appropriate software part naming or classification scheme, (b) collecting or building reusable software components, (c) configuring or composing components into a viable application [1], and (d) maintaining and searching a components library [93]. In turn, each of these dilemmas is mitigated or resolved in practice through the selection of software component granularity.

The granularity of the components (i.e., size, complexity, functional capability) varies greatly across different approaches. Most approaches attempt to utilize components similar to common (textbook) data structures with algorithms for their manipulation: small-grain components. However, the use/reuse of small-grain components in and of itself does not constitute a distinct approach to software evolution. Other approaches attempt to utilize components resembling functionally complete systems or subsystems (e.g., user interface management system): large-grain components. The use/reuse of large-grain components guided by an application domain analysis and subsequent mapping of attributed domain objects and operations onto interrelated components does appear to be an alternative approach to developing software systems [66], and thus is an area of active research.

There are many ways to utilize reusable software components in evolving software systems. However, the cited studies suggest their initial use during architectural or component design specification as a way to speed implementation. They might also be used for prototyping purposes if a suitable software prototyping technology is available.

3.3 APPLICATION GENERATION

Application generation is an approach to software development similar to reuse of parameterized, large-grain software source code components. Such components are configured and specialized to an application domain via a formalized specification language used as input to the application generator. Common examples provide standardized interfaces to database management system applications, and include generators for reports, graphics, user interfaces, and application-specific editors [43].

Application generators give rise to a model of software evolution whereby traditional software design activities are either all but eliminated, or reduced to a data base design problem. The software design activities are eliminated or reduced because the application generator embodies or provides a generic software design that is supposed to be compatible with the application domain. However, users of application generators are usually expected to provide input specifications and application maintenance services. These capabilities are possible since the generators can usually only produce software systems specific to a small number of similar application domains, and usually those that depend on a data base management system.

3.4 SOFTWARE DOCUMENTATION SUPPORT ENVIRONMENTS

Much of the focus on developing software products draws attention to the tangible software artifacts that result. Most often, these products take the form of documents: commented source code listings, structured design diagrams, unit development folders, etc. These documents characterize what the developed system is supposed to do, how it does it, how it was developed, how it was put together and validated, and how to install, use, and maintain it. Thus, a collection of software documents records the passage of a developed software system through a set of life cycle stages.

It seems reasonable that there will be models of software development that focus attention to the systematic production, organization, and management of the software development documents. Further, as *documents are tangible* products, it is common practice when software systems are developed under contract to a private firm, that the delivery of these documents is a contractual stipulation, as well as the basis for receiving payment for development work already performed. Thus, the need to support and validate conformance of these documents to software development and quality assurance standards emerges. However, software development documents are often a primary medium for communication between developers, users, and maintainers that spans organizational space and time. Thus, each of these groups can benefit from automated mechanisms that allow them to browse, query, retrieve, and selectively print documents [35]. As such, we should not be surprised to see construction and deployment of software environments that provide ever increasing automated support for engineering the software documentation life cycle [69, 42, 25, 35].

3.5 PROGRAM EVOLUTION MODELS

In contrast to the preceding four prescriptive product development models, Lehman and Belady sought to develop a descriptive model of software product evolution. They conducted a series of empirical studies of the evolution of large software systems at IBM during the 1970's [59]. Based on their investigations, they identify five properties that characterize the evolution of large software systems. These are:

1. Continuing change: a large software system undergoes continuing change or becomes progressively less useful
2. Increasing complexity: as a software system evolves, its increases unless work is done to maintain or reduce it
3. Fundamental law of program evolution: program evolution, the programming process, and global measures of project and system attributes are statistically self-regulating with determinable trends and invariants
4. Invariant work rate: the rate of global activity in a large software project is statistically invariant
5. Incremental growth limit: during the active life of a large program, the volume of modifications made to successive releases is statistically invariant.

However, it is important to observe that these are global properties of large software systems, not causal mechanisms of software evolution.

4.0 SOFTWARE PRODUCTION PROCESS MODELS

There are two kinds of software production process models: non-operational and operational. Both are software process models. The difference between the two primarily stems from the fact that the operational models can be viewed as computational scripts or programs: programs that implement a particular regimen of software engineering and evolution. Non-operational models on the other hand denote conceptual approaches that have not yet been sufficiently articulated in a form suitable for codification or automated processing.

4.1 NON-OPERATIONAL PROCESS MODELS

There are two classes of non-operational software process models of the great interest. These are the spiral model and the continuous transformation models. There are also a wide selection of other non-operational models which for brevity we label as miscellaneous models. Each is examined in turn.

4.1.1 The Spiral Model

The spiral model of software development and evolution represents a risk-driven approach to software process analysis and structuring [21]. This approach, developed by Barry Boehm, incorporates elements of specification-driven, prototype-driven process methods, together with the classic software life cycle. It does so by representing iterative development cycles as an expanding spiral, with inner cycles denoting early system analysis and prototyping, and outer cycles denoting the classic software life cycle. The radial dimension denotes cumulative development costs, and the angular dimension denotes progress made in accomplishing each development spiral (see figure 3).

Risk analysis, which seeks to identify situations which might cause a development effort to fail or go over budget/schedule, occurs during each spiral cycle. In each cycle, it represents roughly the same amount of angular displacement, while the displaced sweep volume denotes increasing levels of effort required for risk analysis. System development in this model therefore spirals out only so far as needed according to the risk that must be managed. Alternatively, the spiral model indicates that the classic software life cycle model need only be followed when risks are greatest, and after early system prototyping as a way of reducing these risks, albeit at increased cost. Finally, efforts are now in progress to prototype and develop operational versions of the Spiral Model [83].

4.1.2 Continuous Transformation Models

These models propose a process whereby software systems are developed through an ongoing series of transformations of problem statements into abstract specifications into concrete implementations [91, 8, 12, 2]. Lehman, Stenning, and Turski, for example, propose a scheme whereby there is no traditional life cycle nor separate stages, but instead an ongoing series of reifying transformations that turn abstract specifications into more concrete programs [57, 58]. In this sense then, problem statements and software systems can emerge somewhat together, and thus can continue to co-evolve.

Continuous transformation models also accommodate the interests of software formalists who seek the precise statement of formal properties of software system specifications. Accordingly, the specified formalisms can be mathematically transformed into properties that a source implementation should satisfy. The potential for automating such models is apparent, but it still the subject of ongoing research (and addressed below).

4.1.3 Miscellaneous Process Models

Many variations of the non-operational life cycle and process models have been proposed, and appear in the proceedings of the four software process workshops [71, 88, 32, 83]. These include fully interconnected life cycle models which accommodate transitions between any two phases subject to satisfaction of their pre- and post-conditions, as well as compound variations on the traditional life cycle and continuous transformation models. However, the cited reports indicate that in general most software process models are exploratory, so little experience with these models has been reported.

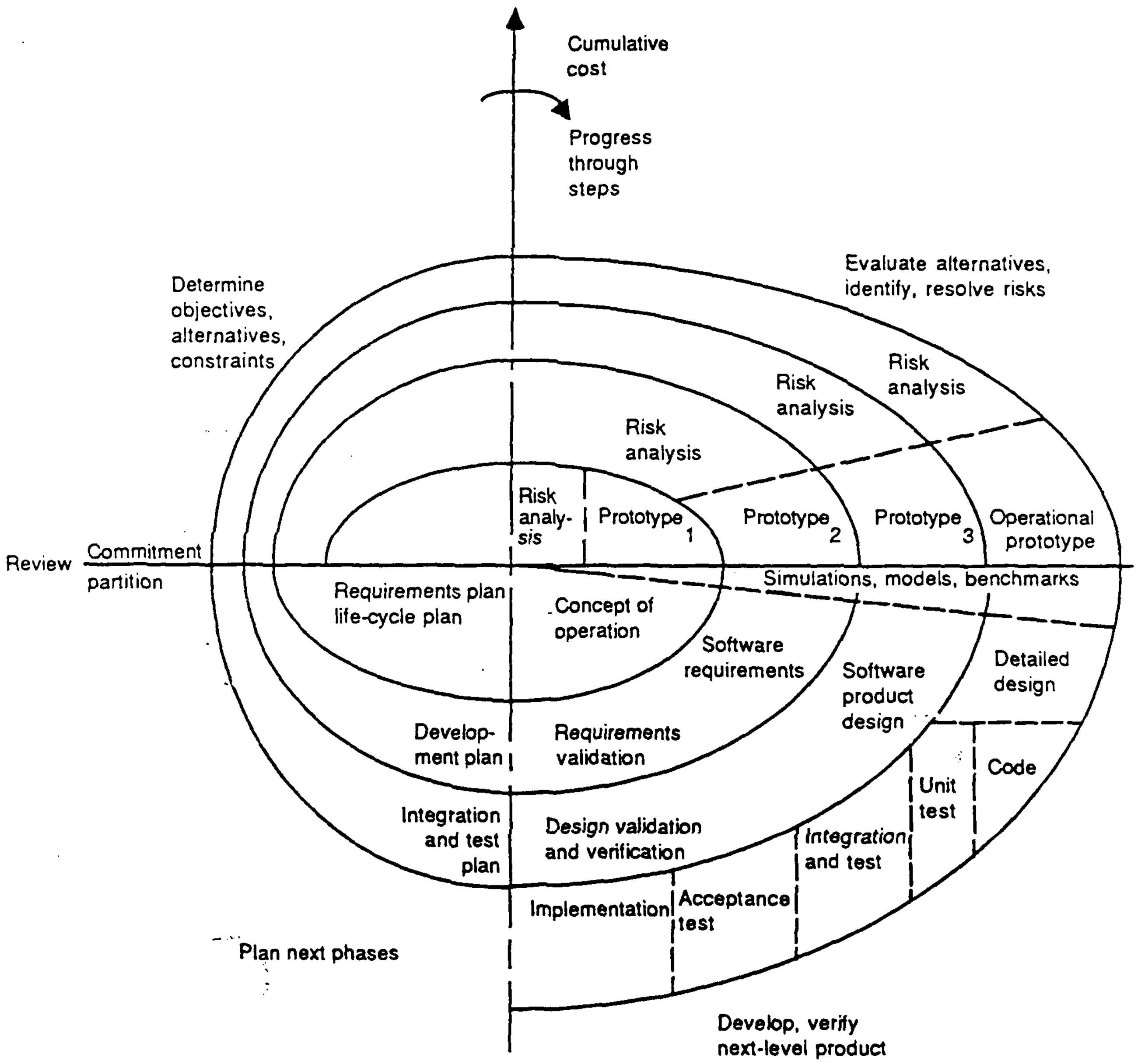

Figure 3. The spiral model diagram.

4.2 OPERATIONAL PROCESS MODELS

In contrast to the preceding non-operational process models, many models are now beginning to appear that codify software engineering processes in computational terms—as programs or executable models. Three classes of operational software process models can be identified and examined.

4.2.1 Operational Specifications For Rapid Prototyping

The operational approach to software development assumes the existence of a formal specification language and processing environment [12, 3, 4, 95]. Specifications in the language are "coded," and when computationally evaluated, constitute a functional prototype of the specified system. When such specifications can be developed and processed incrementally, the resulting system prototypes can be refined and evolved into functionally more complete systems. However, the emerging software systems are always operational in some form during their development. Variations within this approach represent either efforts where the prototype is the end sought, or where specified prototypes are kept operational but refined into a complete system.

The power underlying operational specification technology is determined by the specification language. Simply stated, if the specification language is a conventional programming language, then nothing new in the way of software development is realized. However, if the specification incorporates (or extends to) syntactic and semantic language constructs that are specific to the application domain, which usually are not part of conventional programming languages, then domain-specific rapid prototyping can be supported.

An interesting twist worth note is that it is generally within the capabilities of many operational specification languages to specify "systems" whose purpose is to serve as a model of an arbitrary abstract process, such as a software process model. In this way, using a prototyping language and environment, one might be able to specify an abstract model of some software engineering processes as a system which produces and consumes certain types of documents, as well as the classes of development transformations applied to them. Thus in this regard, it may be possible to construct operational software process models that can be executed or simulated using software prototyping technology. Humphrey and Kellner describe one such application and give an example using the graphic-based state-machine notation provided in the STATECHARTS environment [46].

4.2.2 Software Process Automation And Programming

Process automation and programming are concerned with developing "formal" specifications of how a (family of) software system(s) should be developed. Such specifications therefore should provide an account for the organization and description of various software production task chains, how they interrelate, when they can iterate, etc., as well as what software tools to use to support different tasks, and how these tools should be used [41, 48, 67]. Focus then converges on characterizing the constructs incorporated into the language for specifying and programming software processes. Accordingly, discussion then turns to examine the appropriateness of language constructs for expressing rules for backward and forward-chaining [50], behavior [89], object type structures, process dynamism, constraints, goals, policies, modes of user interaction, plans, off-line activities, resource commitments, etc., across various levels of granularity. This in turn implies that conventional mechanisms such as operating system shell scripts (e.g., Makefiles on Unix) do not support the kinds of software process automation these constructs portend.

Lehman [61] and Curtis et al. [28] provide provocative critiques of the potential and limitations of current proposals for software process automation and programming. Their criticisms, given our framework, essentially point out that many process programming proposals (as of 1987) were focused almost exclusively to those aspects of software engineering that were amenable to automation, such as tool sequence invocation. They point out how such proposals often fail to address how the production settings and products constrain and interact with how the software production process is defined and performed, as revealed in recent empirical software process studies [14, 29, 13].

4.2.3 Knowledge-Based Software Automation (KBSA)

KBSA attempts to take process automation to its limits by assuming that process specifications can be used directly to develop software systems, and to configure development environments to support the production tasks at hand. The common approach is to seek to automate the continuous transformation model [12]. In turn, this implies an automated environment capable of recording the formalized development of operational specifications, successively transforming and refining these specifications into an implemented system, assimilating maintenance requests by incorporating the new/enhanced specifications into the current development derivation, then replaying the revised development toward implementation [5, 6]. However, current progress has been limited to demonstrating such mechanisms and specifications on software coding, maintenance, project communication and management tasks [5, 6, 24, 70, 51, 75, 76], as well as more recently to software component catalogs and formal models of software development processes [68, 93].

5.0 SOFTWARE PRODUCTION SETTING MODELS

In contrast to product or production process models of software evolution, production setting models draw attention to organizational and management strategies for developing and evolving software systems. With rare exception, such models are non-operational. As such, the focus is more strategic. But it should become clear that such strategies do affect what software products get developed, and how software production processes will be organized and performed.

The settings of software evolution can be modeled in terms of the people or programs ("agents") who perform production processes with available resources to produce software products. These agents can play single or multiple roles during a software development effort. Further, their role might be determined by their availability, participation in other organized roles, security access rights, or authority (expertise). A role represents the set of skills (i.e., reliable and robust operational plan) needed to perform some software production task. We often find, for example, software developers in the role(s) of "specification analyst," "coder," or "QA manager." Further, in order for an agent in a particular role to perform her/his task, then a minimum set (configuration) of resources (including tools) and product/process requirements must be provided for task completion. Once again, the descriptions of which agents play which role in performing what tasks with what resources can also be modeled as interrelated attributed objects that can be created, composed, and managed in ways similar to those for software products and processes.

5.1 SOFTWARE PROJECT MANAGEMENT PROCESS MODELS

In parallel to (or on top of) a software development effort, there is normally a management superstructure to configure and orchestrate the effort. This structure represents a cycle of activities for which project managers assume the responsibility. The activities include project planning, budgeting and controlling resources, staffing, dividing and coordinating staff, scheduling deliverables, directing and evaluating (measuring) progress, and intervening to resolve conflicts, breakdowns, or resource distribution anomalies [85, 77, 51, 72, 47]. Planning and control are often emphasized as the critical activities in software project management [85]. However, other studies suggest that division of labor, staffing, coordination, scheduling, intervention (e.g., "putting out fires") and negotiation for additional resources substantially determine the effectiveness of extant planning and control mechanisms as well as product quality and process productivity [77, 36, 14, 28].

5.2 ORGANIZATIONAL SOFTWARE DEVELOPMENT MODELS

Software development projects are plagued with many recurring organizational dilemmas which can slow progress [55, 53, 56, 77, 36, 28, 63]. Problems emerge from unexpected breakdowns or incomplete understandings of the interdependencies that exist between the software products under development, the production techniques in use, and the interactions between the different agents (customers, end users, project managers, staff engineers, maintenance vendors, etc.) and the different resources they transact in the organizational production setting. Experienced managers recognize these dilemmas and develop strategies for mitigating or resolving their adverse effects. Such strategies form an informal model for how to manage software development throughout its life cycle. However, these models or strategies often do not get written down except by occasional outside observers. This of course increases the value of an experienced manager, while leaving the more common unexperienced software development manager disadvantaged. These disadvantaged and less-experienced software managers are therefore more likely to waste organizational effort and scarce resources in recreating, experiencing, and evaluating problems of a similar kind that have already transpired before. In a similar manner, there are key software engineers, "system gurus," and the like who maintain a deep understanding of software system architecture or subtle operational features when such characteristics are not written down [29]. These key people often provide highly-valued expertise, whose loss can be very detrimental to their host organization. Thus, these outside analysts can provide a valuable service through publication of their findings of observed or synthesized heuristics concerning organizational software development dynamics [77, 29].

In another study of software project teamwork in organizational settings, recent research has revealed that (a) there are many different forms of teamwork structure, (b) that software people frequently change their work structure in response to unplanned contingencies, and (c) different patterns of work structures are associated with higher software productivity or higher quality products [13]. What this suggests is that it is possible for organizations to create or impose software production processes which may work well in certain circumstances, but do poorly or fail in others. This suggests that generic software production processes and supporting tools that fail to address how software people will (re)structure their work are only partial solutions. Thus, what is needed is an approach to software development that addresses on more equal terms, the products, production processes, and production settings where people work together to develop, use, and evolve software systems.

5.3 CUSTOMER RESOURCE LIFE CYCLE MODELS

With the help of information (i.e., software) systems, a company can become more competitive in all phases of its customer relationships. The customer resource life cycle (CRLC) model is claimed to make it possible for such companies to determine when opportunities exist for strategic applications [49, 92]. Such applications change a firm's product line or the way a firm competes in its industry. The CRLC model also indicates what specific application systems should be developed.

The CRLC model is based on the following premises: the products that an organization provides to its customers are, from the customer viewpoint, supporting resources. A customer then goes through a cycle of resource definition, adoption, implementation and use. This can require a substantial investment in time, effort, and management attention. But if the supplier organization can assist the customer in managing this resource life cycle, the supplier may then be able to differentiate itself from its competitors via enhanced customer service or direct cost savings. Thus, the supplier organization should seek to develop and apply software systems that support the customer's resource life cycle. [49] and [92] describe two approaches for articulating CRLC models and identifying strategic software system applications to support them.

The purpose of examining such models is to observe that forces and opportunities in a marketplace such as customer relationships, corporate strategy, and competitive advantage can help determine the evolution of certain kinds of software systems.

5.4 SOFTWARE TECHNOLOGY TRANSFER AND TRANSITION MODELS

The software innovation life cycle circumscribes the technological and organizational passage of software systems. This life cycle therefore includes the activities that represent the transfer and transition of a software system from its producers to its consumers. This life cycle includes the following non-linear sequence of activities [73, 79]:

- Invention and prototyping: software research and exploratory prototyping
- Product development: the software development life cycle
- Diffusion: packaging and marketing systems in a form suitable for widespread dissemination and use
- Adoption and Acquisition: deciding to commit organizational resources to get new systems installed
- Implementation: actions performed to assimilate newly acquired systems into existing work and computing arrangements
- Routinization: using implemented systems in ways that seem inevitable and part of standard procedures
- Evolution: sustaining the equilibrium of routine use for systems embedded in community of organizational settings through enhancements, restructuring, debugging, conversions, and replacements with newer systems.

Available research indicates that progress through the software innovation life cycle can take 7-20 years for major software technologies (e.g., Unix, expert systems, programming environments, Ada) [73]. Thus, moving a software development organization to a new technology can take a long time, great effort, and many perceived high risks. Research also indicates that most software innovations (small or large) fail to get properly implemented, and thus result in wasted effort and resources [79]. The failure here is generally not technical, but instead primarily organizational. Thus, organizational circumstances and the people who animate them have far greater affect in determining the successful use and evolution of a software innovation, than the innovation's technical merit. However, software technology transfer is an area requiring much more research [79].

5.5 OTHER MODELS OF SYSTEM PRODUCTION AND MANUFACTURING

What other kinds of models of software production might be possible? If we look to see how other technological systems are developed, we find the following sort of models for system production:

- Ad-hoc problem solving, tinkering, and articulation work: the weakest model of production is when people approach a development effort with little or no prepared plan at hand, and thus rely solely upon their skill, ad hoc tools, or the loosely coordinated efforts of others to get them through. It is situation specific and driven by accommodations to local circumstances. It is therefore perhaps the most widely practiced form of production and system evolution.

- Group project: software life cycle and process efforts are usually realized one at a time, with every system being treated somewhat uniquely. Thus such efforts are often organized as group projects.

- Custom job shop: job shops take on only particular kinds of group project work, due to more substantial investment in tooling and production skill/technique refinement as well as more articulate application requirements. Most software development organizations, whether an independent firm or a unit of a larger firm, are operated as software job shops.

- Batched production: provides the customization of job shops but for a larger production volume. Subsystems in development are configured on jigs that can either be brought to workers and production tools, or that tools and workers can be brought to the raw materials for manufacturing or fabrication or to the subsystems.

- Pipeline: when system development requires the customization of job shops or the specialization of volume of batched production, while at the same time allowing for concurrently staged sequences of subsystem development. The construction of tract housing and office towers are typically built according to, respectively, horizontal or vertical pipelines.

- Flexible manufacturing systems: seek to provide the customization capabilities of job shops, while relying upon advanced automation to allow economies of scale, task standardization, and delivery of workpieces of transfers lines realized through rapidly reconfigurable workstation tooling and process programming. Recent proposals for "software factories" have adopted a variation of this model called flexible software manufacturing systems [81].

- Transfer (assembly) lines: when raw input resources or semi-finished sub-assemblies can be moved through a network of single action manufacturing workstations. Such a production network is called either a transfer line or assembly line. Most mass-produced consumer goods (those with high production volumes and low product variation) are manufactured on some form of assembly line. Notions of productivity measurement and quality assurance are most often associated with, and most easily applied to, transfer/assembly lines.

- Continuous process control: when the rate or volume of uniform raw input resources and finished output products can be made continuous and automatically variable, then a continuous process control form of production is appropriate. Oil refining is an example of such a process, with crude oil from wells as input, and petroleum products (gasoline, kerosene, multi-grade motor oil) as outputs. Whether software can be produced in such a manner is unlikely at this time.

6.0 WHERE DO TOOLS AND TECHNIQUES FIT INTO THE MODELS?

Given the diversity of software life cycle and process models, where do software engineering tools and techniques fit into the picture? This section briefly identifies some of the places where different software engineering technologies can be matched to certain models. Another way to look at this section might be to consider instead what software engineering technologies might be available in your setting, then seek a model of software evolution that is compatible.

6.1 LIFE CYCLE SUPPORT MECHANISMS

Most of the traditional life cycle models are decomposed as stages. These stages then provide boundaries whereby software engineering technologies are targeted. Thus, we find engineering techniques or methods (e.g., Yourdon structured design, TRW's software requirements engineering methodology (SREM)) being targeted to support different life cycle stages, and tools (e.g., TRW's requirements engineering and verification system (REVS)) targeted to support the associated activities. However, there are very few, if any, packages of tools and techniques that purport to provide integrated support for engineering software systems throughout their life cycle [81]. Perhaps this is a shortcoming of the traditional models, perhaps indicative that the integration required is too substantial to justify its expected costs or benefits, or perhaps the necessary technology is still in its infancy. Thus, at present, we are more likely to find ad-hoc or loose collections of software engineering tools and techniques that provide partial support for software life cycle engineering.

6.2 PROCESS SUPPORT MECHANISMS

There are at least three kinds of software process support mechanisms: product articulation technologies, process measurement and analysis technologies, and computational process models and environments.

Product articulation technologies denote the software prototyping tools, reusable software components libraries, and application generator languages, and documentation support environments for more rapidly developing new software systems. These technologies often embody or implicitly support a software product development life cycle when restricted to well-chosen and narrow application domains. This means that these technologies can be employed in ways that enable the traditional software life cycle stages to be performed with varying degrees of automated assistance.

Process measurement and analysis technologies denote the questionnaire, survey, or performance monitoring instruments used to collect quantifiable data on the evolving characteristics of software products, processes, and settings. Collected data can in turn be analyzed with statistical packages to determine descriptive and inferential relationships within the data. These relationships can then be interpreted as indicators for where to make changes in current practices through a restructuring of work/resources, or through the introduction of new software engineering technologies. Such measurement and analysis technologies can therefore accommodate process refinements that improve its overall performance and product quality.

Computational process models denote formalized descriptions of software development activities in a form suitable for automated processing. Such models are envisioned to eventually be strongly coupled to available process support environments which supports the configuration and use of software engineering tools and techniques to be programmed and enacted. These models will be knowledge-intensive: that is, they will incorporate extensive knowledge of the characteristics and interrelationships between software development products, processes, and production settings. In turn, these models will be described in languages whose constructs and formalization may go well beyond those found in popular programming languages or simulation systems. However, at present, computational process models primarily serve to help articulate more precise descriptions for how to conduct different software engineering activities, while programmable support environments are still in the early stages of research.

7.0 EVALUATING LIFE CYCLE MODELS AND METHODOLOGIES

Given the diversity of software life cycle and process models, how do we decide which if any is best, or which to follow? Answering this question requires further research in general, and knowledge of specific software development efforts in particular.

7.1 COMPARATIVE EVALUATION OF LIFE CYCLE AND PROCESS METHODOLOGIES

As noted in Section I, descriptive models of software evolution require the empirical study of its products. how and where they are produced as well as their interdependencies. Therefore, how should such a study be designed to realize useful, generalizable results?

Basically, empirical studies of actual software life cycles or processes should ultimately lead to models of evolution with testable predictions [30, 10]. Such models in turn must account for the dynamics of software evolution across different types of application programs, engineering techniques, and production settings across different sets of comparable data. This means that studies of software evolution must utilize measurements that are reliable, valid, and stable. Reliability refers to the extent that the measures are accurate and repeatable. Validity indicates whether the measured values of process variables are in fact correct. Stability denotes that the instrument measures one or more process variables in a consistent manner across different data sets [30]. Constraints such as these thus usually point to the need for research methods and instruments that give rise to quantitative or statistical results [30, 9, 10].

However, most statistical instruments are geared for snapshot studies where certain variables can be controlled, while others are independent. Lehman and Belady utilize such instruments in their evaluation of large software system attributes [59]. Their study utilizes data collected over periodic intervals for a sample of large software systems over a number of years. However, their results only make strong predictions about global dynamics of product (program) evolution. That is, they cannot predict what will happen at different life cycle stages, in different circumstances, or for different kinds of software systems. To make such predictions requires a different kind of study, analysis and instrumentation.

Van den Bosch, et al. [87], and Curtis et al. [28, 29] among others [14, 13] propose an alternative approach to studying software evolution. They rely upon comparative field studies of a sample of software efforts in different organizational settings. Their approach is targeted to constructing a framework for discovering the mechanisms and organizational processes that shape software evolution with a comparative study sample. The generality of the results they derive can thus be assessed in terms of the representativeness of their sample space.

In a different way, Kelly [52] provides an informing comparative analysis of four methods for the design of real-time software systems. Although his investigation does not compare models of software evolution, his framework is suggestive of what might be accomplished through comparative analysis of such models.

Other approaches that report on the comparative analysis of software evolution activities and outcomes can be found elsewhere [55, 9, 19].

7.2 RESEARCH PROBLEMS AND OPPORTUNITIES

As should be apparent, most of the alternative models of software evolution are relatively new, and in need of improv⁻ment and empirical grounding. It should however also be clear that such matters require research investigations. Prescriptive models can be easy to come by, whereas descriptive models require systematic research regimens which can be costly, but of potentially higher quality and utility. Nonetheless, there are many opportunities to further develop, combine, or refute any of the alternative models of software evolution. Comparative research design methods, data sampling, collection, and analysis are all critical topics that require careful articulation and scrutiny [10]. And each of the alternative models, whether focussing attention to either software products, production processes, production settings, or their combination can ideally draw upon descriptive studies as the basis of their prescriptions [7]. In turn, such models and studies will require notations or languages whose constructs support computational formalization, analysis, and processing. This processing is needed to insure the consistency, completeness, and traceability of the modeled processes, as well as to provide a host environment of conducting experimental or improvement-oriented studies of software evolution. Thus, we are at a point where empirical studies of software life cycle and process models (or their components) are needed, and likely to be very influential if investigated systematically and rigorously.

Therefore, in broader terms, it is appropriate to devote some attention to the problem of designing a set of experiments intended to substantiate or refute a model of software evolution, where critical attention should then be devoted to evaluating the quality and practicality (i.e., time, effort, and resources required) of the proposed research.

8.0 CUSTOMIZABLE LIFE CYCLE PROCESS MODELS

Given the emerging plethora of models of software evolution, how does one choose which model to put into practice? This will be a recurring question in the absence of empirical support for the value of one model over others. We can choose whether to select an existing model, or else to develop a custom model. Either way, the purpose of having a model is to use it to organize software development efforts in a more effective, more productive way. But this is not a one-shot undertaking. Instead, a model of software evolution is likely to be most informing when not only used to prescribe software development organization, but also when used to continually measure, tune, and refine the organization to be more productive, risk-reducing, and quality driven [47, 72, 11]. In this regard, the purpose in selecting, defining, and applying a model of software evolution is to determine what, how, and where to intervene or change current software production practices or policies.

8.1 SELECTING AN EXISTING MODEL

Choosing the model that's right for your software project and organization is the basic concern. At this time, we can make no specific recommendation for which model is best in all circumstances. The choice is therefore open-ended. However, we might expect to see the following kinds of choices being made with respect to existing models: At present, most software development organizations are likely to adopt one of the traditional life cycle models. Then they will act to customize it to be compatible with other organizational policies, procedures, and market conditions. Software research organizations will more likely adopt an alternative model, since they are likely to be interested in evaluating the potential of emerging software technologies. When development organizations adopt software technologies more closely aligned to the alternative models (e.g., reusable components, rapid prototyping), they may try to use such models either experimentally, or to shoehorn them into a traditional life cycle model, with many evolutionary activities kept informal and undocumented. Alternatively, another strategy to follow is to do what some similar organization has done, and to use the model they employ. Studies published by researchers at IBM and AT&T Bell Laboratories are often influential in this regard [47, 72, 94].

8.2 CUSTOMIZING YOUR OWN MODEL

Basili and Rohmbach [11] are among those who advocate the development of a custom life cycle process model for each project and organization. Empirical studies of software development seem to indicate that life cycle process modeling will be most effective and have the greatest benefit if practiced as a regular activity. Process metrics and measurements need to be regularly applied to capture data on the effectiveness of current process activities. As suggested above, it seems likely that at this time, the conservative strategy will be to adopt a traditional life cycle model and then seek to modify or extend it to accommodate new software product or production process technologies and measures. However, it seems just as likely that software development efforts that adopt software product, production process and production setting concerns into a comprehensive model may have the greatest potential for realizing substantial improvement in software productivity, quality, and cost reduction [80].

8.3 USING PROCESS METRICS AND EMPIRICAL MEASUREMENTS

One important purpose of building or buying a process model is to be able to apply it to current software development projects in order to improve project productivity, quality, and cost-effectiveness [47, 72]. The models therefore provide a basis for instrumenting the software process in ways that potentially reveal where development activities are less effective, where resource bottlenecks occur, and where management interventions or new technologies could have a beneficial impact [11, 94]. Scacchi and Kintala [80] go so far as to advocate an approach involving the application of knowledge-based technologies for modeling and simulating software product, production process, and production setting interactions based upon empirical data (i.e., knowledge) acquired through questionnaire surveys, staff interviews, observations, and on-line monitoring systems. Such an approach is clearly within the realm of basic research, but perhaps indicative of the interest in developing high-potential, customizable models of software evolution.

8.4 STAFFING THE LIFE CYCLE PROCESS MODELING ACTIVITY

Ideally, the staff candidate best equipped to organize or analyze an organization's model of software evolution is one who has mastered the range of material outlined above, or along the lines suggested elsewhere [45]. That is, a staff member who has only had an introductory or even intermediate level exposure to this material is not likely to perform software life cycle or process modeling competently. Large software development organizations with dozens, hundreds, or even thousands of software developers are likely to rely upon one or more staff members with a reasonably strong background in local software development practices and experimental research skills. This suggests that such staff are therefore likely to possess the equivalent of a masters or doctoral degree software engineering or experimental computer science. In particular, a strong familiarity with experimental research methods, field studies, sampling strategies, questionnaire design, survey analysis, statistical data analysis package. and emerging software technologies are the appropriate prerequisites. Simply put, this is not a job for any software engineer, but instead a job for software engineer (or industrial scientist) with advanced training and experience in experimental research tools and techniques.

9.0 CONCLUSIONS

In conclusion, we reiterate our position: contemporary models of software evolution must account for the products, production processes, and settings, as well as their interrelationships that arise during software development, use, and maintenance. Such models can therefore utilize features of traditional software life cycle models, as well as those of automatable software process models.

The models of software evolution that were presented and compared in this article begin to suggest the basis for establishing a substantial theory of software evolution. Such a theory, however, will not likely take on a simple form. Software evolution is a "soft and difficult" research problem whose manifestation lies in complex organizational settings that are themselves open-ended and evolving. This suggests that significant attempts to formulate such a theory must be based upon comparative analysis of systematic empirical observations of software development efforts across multiple cases. Prescriptive models of software evolution that lack such grounding can at best be innovative and insightful, but nonetheless speculative. Prescriptive models might in fact give rise to new technological regimens and artifacts, and thereby become yet another force that shapes how software systems evolve. Thus, there *is an important role to be served by prescriptive models, but such a role is not the same as that* intended for a descriptive, empirically grounded theory with testable predicted outcomes.

A number of potential areas for further research were identified as well. First, there are tools and techniques in domains such as manufacturing and chemical engineering which have been automated and in use for a number of years with observable results. This suggests that these technologies should be investigated to determine whether or how their (prescriptive) conceptual mechanisms can be lifted and transformed for use in software system development projects. Similarly, as models of software evolution grow to include software technology transfer and transition, there will be need for tools and techniques to support the capture, analysis, and processing of the enlarged models as well.

Second, there is a unproductive and ineffective divide between (a) the technology-driven software product and production process models, and (b) the organization-driven software production setting models. Recent research efforts have begun to cover the gap through the utilization of descriptive analytical frameworks that treat technological and organizational variables in equal, interdependent terms. Continuing investigations along this line should lead to more substantive models or theories software evolution than continued efforts on only one side of the divide.

Third, given new models or theories of software evolution, we can again address questions of where software engineering tools and techniques fit into the models. Conversely, when new software development technologies become available, how will they mesh with the new models? In any event, it becomes clear that the models of software evolution must themselves evolve, or else become dated and potentially another reverse salient in software engineering progress.

Fourth, the comparative, empirical evaluation of software development efforts and models of evolution is still an uncommon tradition in the software engineering research community. Perhaps this is due to the relatively small number of software engineering people who are trained in the requisite qualitative and quantitative methods that are part of the experimental research design tradition [30, 9, 19, 87, 20, 10, 14, 29, 13]. Therefore, this represents an area that should be addressed through software engineering curriculum enhancements. Nonetheless, their is a dearth of systematic empirical studies of actual software development, use, and maintenance efforts. Thus, there is a clear need to support the growth of this kind of science. Last, both of these recommendations represent an opportunity to create a national archive of experimental studies, data, and results whose network-accessible database would be treated as a national resource that would continue to grow over time if proven effective.

Fifth, the potential to develop models of software evolution that can be customized to product, production process, and production setting characteristics holds promise for a tangible payoff from the modeling efforts. Such models offer the potential to address both the technological and organizational aspects of software evolution. Further, such models will benefit from explicit computational formulation and realization. This in turn suggests that new software engineering environments may be constructed that can directly capture, model and simulate—literally execute software development in an abstract form as a way of running a software development effort ahead of itself. Such simulation together with empirical observation provides a means for refining and improving the model of software evolution, as well as serving as a growing archive of local software development history and experience.

Overall, the goal of developing an empirically grounded theory of software evolution is ambitious. Building models of software evolution in computational form is an increasingly inevitable requirement for making progress in this area. This paper therefore serves as a starting point and an initial map for how to get there.

10.0 ACKNOWLEDGEMENTS

This report incorporates and extends material originally appeared as a curriculum module (CM-10.1-87) prepared by the author at the Software Engineering Education Division of the Software Engineering Institute at Carnegie-Mellon University, Pittsburgh, PA. Many helpful and clarifying comments were provided by Priscilla Fowler and Marc Kellner of SEI, and Robert Glass. In addition, reviews and comments were provided by Salah Bendifallah, Song Choi, and Pankaj Garg at USC. At USC, this research has been supported by contracts and grants from AT&T Bell Laboratories, Bell Communications Research, Eastman Kodak Inc., Hughes Aircraft Radar Systems Group, Office of Naval Technology through the Naval Ocean Systems Center, and Pacific Bell.

11.0 REFERENCES

1. B.P. Allen and S.D. Lee. A Knowledge-based Environment for the Development of Software Parts Composition Systems. *Proc. 11th. Intern. Conf. Software Engineering, IEEE Computer Society, 1989*, pp. 104-112.

2. Balzer, R. "Transformational Implementation: An Example". *IEEE Trans. Software Engineering SE-7*, 1 (1981), 3-14.

3. Balzer, R., N. Goldman, and D. Wile. "Operational Specifications as the Basis for Rapid Prototyping". ACM Software Engineering Notes 7, 5 (1982), 3-16.

4. Balzer, R., D. Cohen, M. Feather, N. Goldman W. Swartout, and D. Wile. Operational Specifications as the Basis for Specification Validation. In *Theory and Practice of Software Technology, North-Holland*, Amsterdam, 1983.

5. Balzer, R., T. Cheatham, and C. Green. "Software Technology in the 1990's: Using a New Paradigm". *Computer* 16, 11 (Nov. 1983), 39-46.

6. Balzer, R. "A 15 Year Perspective on Automatic Programming". *IEEE Trans. Software Engineering SE-11*, 11 (Nov. 1985), 1257-1267.

7. Basili, V.R. Software Development: A Paradigm for the Future. *Proc. COMPSAC 89, IEEE Computer Society*, 1989, pp. 3-15.

8. Basili, V.R., and A.J. Turner. "Iterative Enhancement: A Practical Technique for Software Development". *IEEE Trans. Software Engineering SE-1*, 4 (Dec. 1975), 390-396.

9. Basili, V.R., and R.W. Reiter. "A Controlled Experiment Quantitatively Comparing Software Development Approaches". *IEEE Trans. Software Engineering SE-7*, 3 (May 1981), 299-320.

10. Basili, V.R., R. Selby, and D. Hutchens. "Experimentation in Software Engineering". *IEEE Trans. Software Engineering SE-12*, 7 (July 1986), 733-743.

11. Basili, V.R., and H.D. Rombach. Tailoring the Software Process to Project Goals and Environments. *Proc. 9th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1987, pp. 345-357.

12. Bauer, F.L. Programming as an Evolutionary Process. *Proc. 2nd. Intern. Conf. Software Engineering, IEEE Computer Society*, Jan., 1976, pp. 223-234.

13. Bendifallah, S., and W. Scacchi. Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork. *Proc. 11th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1989, pp. 260-270.

14. Bendifallah, S., and W. Scacchi. "Understanding Software Maintenance Work". *IEEE Trans. Software Engineering SE-13*, 3 (March 1987), 311-323.

15. Benington, H.D. "Production of Large Computer Programs". *Annals of the History of Computing* 5, 4 (1983), 350-361. (Original version appeared in 1956. Also appears in *Proc. 9th. Intern. Conf. Software Engineering*, 299-310).

16. Biggerstaff, T., and A. Perlis (eds.). "Special Issues on Software Reusability". *IEEE Trans. Software Engineering SE-10*, 5 (Sept. 1984).

17. Boehm, B. "Software Engineering". *IEEE Trans. Computer* C-25, 12 (Dec. 1976), 1226-1241.

18. Boehm, B.W. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, N. J., 1981.

19. Boehm, B. "An Experiment in Small-Scale Software Engineering". *IEEE Trans. Software Engineering SE-7*, 5 (Sept. 1981), 482-493.

20. Boehm, B.W., T. Gray, and T. Seewaldt. Prototyping vs. Specifying: A Multi-project Experiment. *Proc. 7th. Intern. Conf. Soft. Engr.*, 1984, pp. 473-484.

21. Boehm, B.W. "A Spiral Model of Software Development and Enhancement". ACM Software Engineering Notes 11, 4 (1986), 22-42.

22. Bowler, P.J. Evolution: The History of an Idea. University of California Press, 1983.

23. Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven. Approaches to Prototyping. Springer-Verlag, New York, 1984.

24. Cheatham, T. Supporting the Software Process. *Proc. 19th. Hawaii Intern. Conf. Systems Sciences*, 1986, pp. 814-821.

25. Choi, S., and W. Scacchi. "Assuring the Correctness of Configured Software Descriptions". *Proc. 2nd. Intern. Workshop Software Configuration Management*. ACM Software Engineering Notes, 17(7) (1989), 67-76.

26. Chikofsky, E.J. and J.H. Cross. "Reverse Engineering and Design Recovery". *IEEE Software 7*, 1 (1990), 13-18. Special Issue on Maintenance, Reverse Engineering and Design Recovery.

27. Connell, J.L., and L.B. Shafer. Structured Rapid Prototyping. Yourdon Press, 1989.

28. Curtis, B., H. Krasner, V. Shen, and N. Iscoe. On Building Software Process Models Under the Lamppost. *Proc. 9th. Intern. Conf. Software Engineering, IEEE Computer Society*, April, 1987, pp. 96-103.

29. Curtis, B., H. Krasner, and N. Iscoe. "A Field Study of the Software Design Process for Large Systems". *Communications ACM 31*, 11 (November 1988), 1268-1287.

30. Curtis, B. "Measurement and Experimentation in Software Engineering". *Proceedings IEEE 68*, 9 (1980), 1144-1157.

31. Distaso, J. "Software Management—A Survey of Practice in 1980". *Proceedings IEEE 68*, 9 (1980), 1103-1119.

32. IEEE Computer Society. 3rd. Intern. Software Process Workshop, Los Alamitos, Calif., 1986.

33. Fairley, R. Software Engineering Concepts. mh, New York, 1985.

34. Garg, P.K. and W. Scacchi. "ISHYS: Design of an Intelligent Software Hypertext Environment". *IEEE Expert 4*, 3 (Fall 1989), 52-63.

35. Garg, P.K. and W. Scacchi. "A Hypertext System to Manage Software Life Cycle Documents". *IEEE Software 7*, 2 (1990), (to appear).

36. Gasser, L. "The Integration of Computing and Routine Work". ACM Trans. Office Information Systems 4, 3 (July 1986), 205-225.

37. Gerson, E. and S.L. Star. "Analyzing Due Process in the Workplace". ACM Trans. Office Info. Sys. 4, 3 (1986), 257-270.

38. Goguen, J. "Reusing and Interconnecting Software Components". *Computer* 19, 2 (Feb. 1986), 16-28.

39. Graham, D.R. "Incremental Development: Review of Nonmonolithic Life-Cycle Development Models". *Information and Software Technology 31*, 1 (January 1989), 7-20.

40. Hekmatpour, S. "Experience with Evolutionary Prototyping in a Large Software Project". ACM Software Engineering Notes 12, 1 (1987), 38-41.

41. Hoffnagel, G.F., and W. Beregi. "Automating the Software Development Process". *IBM Syst. J.* 24, 2 (1985), 102-120.

42. Horowitz, E. and R. Williamson. "SODOS: A Software Documentation Support Environment—Its Definition". *IEEE Trans. Software Engineering SE-12*, 8 (1986), .

43. Horowitz, E., A. Kemper, and B. Narasimhan. "A Survey of Application Generators". *IEEE Software 2*, 1 (Jan. 1985), 40-54.

44. Hosier, W.A. "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming". *IRE Trans. Engineering Management EM-8* (June 1961). (Also appears in *Proc. 9th. Intern. Conf. Software Engineering*, 311-327).

45. Humphrey, W.S. "Characterizing the Software Process". *IEEE Software 5*, 2 (1988), 73-79.

46. Humphrey, W.S. and M. Kellner. Software Process Modeling: Principles of Entity Process Models. *Proc. 11th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1989, pp. 331-342.

47. Humphrey, W.S. "The IBM Large-Systems Software Development Process: Objectives and Direction". *IBM Syst. J.* 24, 2 (1985), 76-78.

48. Huseth, S., and D. Vines. Describing the Software Process. *Proc. 3rd. Intern. Software Process Workshop, IEEE Computer Society*, 1986, pp. 33-35.

49. Ives, B., and G. P. Learmonth. "The Information System as a Competitive Weapon". *Comm. ACM 27*, 12 (Dec. 1984), 1193-1201.

50. Kaiser, G., P. Feiler, and S. Popovich. "Intelligent Assistance for Software Development and Maintenance". *IEEE Software 5*, 3 (1988).

51. Kedzierski, B.I. Knowledge-Based Project Management and Communication Support in a System Development Environment. *Proc. 4th. Jerusalem Conf. Info. Technology*, 1984, pp. 444-451.

52. Kelly, J.C. A Comparison of Four Design Methods for Real-Time Systems. *Proc. 9th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1987, pp. 238-252.

53. Kidder, T. The Soul of a New Machine. Atlantic Monthly Press, New York, 1981.

54. King, J.L., and K.K. Kraemer. "Evolution and Organizational Information Systems: An Assessment of Nolan's Stage Model". *Comm. ACM 27*, 5 (May 1984), 466-475.

55. Kling, R., and W. Scacchi. "Computing as Social Action: The Social Dynamics of Computing in Complex Organizations". *Advances in Computers 19* (1980), 249-327. Academic Press, New York. 56. Kling, R., and W. Scacchi. "The Web of Computing: Computer Technology as Social Organization". *Advances in Computers 21* (1982), 1-90. Academic Press, New York.

57. Lehman, M.M., V. Stenning, and W. Turski. "Another Look at Software Development Methodology". ACM Software Engineering Notes 9, 2 (April 1984), 21-37.

58. Lehman, M.M. A Further Model of Coherent Programming Processes. *Proc. Software Process Workshop, IEEE Computer Society*, 1984, pp. 27-33.

59. Lehman, M.M., and L. Belady. Program Evolution: Processes of Software Change. Academic Press, New York, 1985.

60. Lehman, M.M. Modes of Evolution. *Proc. 3rd. Intern. Software Process Workshop, IEEE Computer Society*, 1986, pp. 29-32.

61. Lehman, M.M. Process Models, Process Programming, Programming Support. *Proc. 9th. Intern. Conf. Software Engineering, IEEE Computer Society*, April, 1987, pp. 14-16.

62. Lientz, B.P. and E.B. Swanson. Software Maintenance Management. Addison-Wesley, 1980.

63. Liker, J.K., and W.M. Hancock. "Organizational Systems Barriers to Engineering Effectiveness". *IEEE Trans. Engineering Management EM-33*, 2 (1986), 82-91.

64. Dept. of Defense. DRAFT Military Standard: Defense System Software Development. DOD-STD-2167A.

65. Mills, H.D., M. Dyer and R.C. Linger. "Cleanroom Software Engineering". *IEEE Software 4*, 5 (1987), 19-25.

66. Neighbors, J. "The Draco Approach to Constructing Software from Reusable Components". *IEEE Trans. Software Engineering SE-10*, 5 (Sept. 1984), 564-574.

67. Osterweil, L. Software Processes are Software Too. *Proc. 9th. Intern. Conf. Software Engineering, IEEE Computer Society*, April, 1987, pp. 2-13.

68. Ould, M.A., and C. Roberts. Defining Formal Models of the Software Development Process. In *Software Engineering Environments*, P. Brererton, Ed., Ellis Horwood, Chichester, England, 1988, pp. 13-26.

69. Penedo, M.H. and E.D. Stuckle. PMDB—A Project Master Database for Software Engineering Environments. *Proc. 8th. Intern. Conf. Soft. Engr., IEEE Computer Society*, 1985, pp. 150-157.

70. Polak, W. Framework for a Knowledge-Based Programming Environment. Workshop on Advanced Programming Environments, Springer-Verlag, 1986.

71. IEEE Computer Society. Software Process Workshop, Los Alamitos, CA, 1984.

72. Radice, R.A., N.K. Roth, A.L. O'Hara, Jr., and W.A. Ciarfella. "A Programming Process Architecture". *IBM Syst. J.* 24, 2 (1985), 79-90.

73. Redwine, S., and W. Riddle. Software Technology Maturation. *Proc. 8th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1985, pp. 189-200.

74. Royce, W.W. Managing the Development of Large Software Systems. *Proc. 9th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1987, pp. 328-338. Originally published in *Proc. WESCON*, 1970.

75. Sathi, A., M. S. Fox, and M. Greenberg. "Representation of Activity Knowledge for Project Management". *IEEE Trans. Patt. Anal. and Mach. Intell. PAMI-7*, 5 (1985), 531-552.

76. Sathi, A., T. Morton, and S. Roth. "Callisto: An Intelligent Project Management System". *AI Magazine* 7, 5 (1986), 34-52.

77. Scacchi, W. "Managing Software Engineering Projects: A Social Analysis". *IEEE Trans. Software Engineering SE-10*, 1 (Jan. 1984), 49-59.

78. Scacchi, W. "Shaping Software Behemoths". *UNIX Review* 4, 10 (Oct. 1986), 46-55.

79. Scacchi, W. and J. Babcock. Understanding Software Technology Transfer. Internal report, Software Technology Program, Microelectronics and Computer Technology Corp., Austin, Texas. (Submitted for publication).

80. Scacchi, W., and C.M.K. Kintala. Understanding Software Productivity. Internal report, Advanced Software Concepts Dept., AT&T Bell Laboratories, Murray Hill, N. J. (Submitted for publication).

81. Scacchi, W. The System Factory Approach to Software Engineering Education. In *Educational Issues in Software Engineering*, Springer-Verlag, New York, 1988. (To appear).

82. Selby, R.W., V.R. Basili, and T. Baker. "CLEANROOM Software Development: An Empirical Evaluation". *IEEE Trans. Soft. Engr. SE-13*, 9 (1987), 1027-1037.

83. ACM Press. Proc. Fourth International Software Process Workshop, Devon, UK, 1988.

84. Squires, S., M. Barnstad, M. Zelkowitz (eds.). "Special Issue on Rapid Prototyping". ACM Software Engineering Notes 7, 5 (Dec. 1982).

85. Thayer, R., A. Pyster, and R. Wood. "Major Issues in Software Engineering Project Management". *IEEE Trans. Software Engineering SE-7*, 4 (July 1981).

86. Tully, C. Software Development Models. Proc. *Software Process Workshop, IEEE Computer Society*, 1984, pp. 37-44.

87. van den Bosch, F., J. Ellis, P. Freeman, L. Johnson, C. McClure, D. Robinson, W. Scacchi, B. Scheft, A. van Staa, and L. Tripp. "Evaluating the Implementation of Software Development Life Cycle Methodologies". ACM Software Engineering Notes 7, 1 (Jan. 1982), 45-61.

88. Wileden, J., and M. Dowson (eds.). "Second Intern. Workshop on Software Process and Software Environments". ACM Software Engineering Notes 11, 4 (1986).

89. Williams, L. Software Process Modeling: A Behavioral Approach. *Proc. 10th. Intern. Conf. Software Engineering, IEEE Computer Society*, 1988, pp. 174-200.

90. T. Winograd and F. Flores. Understanding Computers and Cognition: A New Foundation for Design. Ablex Publishers, 1986.

91. Wirth, N. "Program Development by Stepwise Refinement". *Comm. ACM 14*, 4 (April 1971), 221-227.

92. Wiseman, C. Strategy and Computers: Information Systems as Competitive Weapons. Dow Jones Irwin, New York, 1985.

93. Wood, M., and I. Sommerville. A Knowledge-Based Software Components Catalogue. In *Software Engineering Environments*, P. Brererton, Ed., Ellis Horwood, Chichester, England, 1988, pp. 116-131.

94. Yacobellis, R. H. Software and Development Process Quality Metrics. *Proc. COMPSAC 84, IEEE Computer Society*, 1984, pp. 262-269.

95. Zave, P. "The Operational Versus the Conventional Approach to Software Development". *Comm. ACM 27* (Feb. 1984), 104-118.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE July 1990	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE MODELS OF SOFTWARE EVOLUTION Life Cycle and Process		5. FUNDING NUMBERS C: N66001-87-D-0179 PE: 0602234N WU: DN088690
6. AUTHOR(S)		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California Computer Science Department Los Angeles, CA 90089-0782		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Ocean Systems Center San Diego, CA 92152-5000		10. SPONSORING/MONITORING AGENCY REPORT NUMBER NOSC TD 1893
11. SUPPLEMENTARY NOTES		
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document categorizes and examines a number of schemes for modelling software evolution. The document provides some definitions of the terms used to characterize and compare different models of software evolution.

14. SUBJECT TERMS software evolution life cycle models			15. NUMBER OF PAGES 36
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAME AS REPORT